PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-In-Memory

Yiwei Zhao
Carnegie Mellon University
Pittsburgh, PA, USA
yiweiz3@andrew.cmu.edu

Hongbo Kang
Tsinghua University
Beijing, China
khb20@mails.tsinghua.edu.cn

Ziyang Men
University of California, Riverside
Riverside, CA, USA
zmen002@ucr.edu

Yan Gu
University of California, Riverside
Riverside, CA, USA
ygu@cs.ucr.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Laxman Dhulipala
University of Maryland
College Park, MD, USA
laxman@umd.edu

Charles McGuffey
Reed College
Portland, OR, USA
cmcguffey@reed.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, PA, USA
gibbons@cs.cmu.edu

Abstract

Space-partitioning indexes are widely used for managing multi-dimensional data, but their throughput is often memory-bottlenecked. Processing-in-memory (PIM), an emerging architectural paradigm, mitigates memory bottlenecks by embedding processing cores directly within memory modules, allowing computation to be offloaded to these PIM cores.

In this paper, we present PIM-zd-tree, the first space-partitioning index specifically designed for real-world PIM systems. PIM-zd-tree employs a tunable multi-layer structure, with each layer adopting distinct data layouts, partitioning schemes, and caching strategies. Its design is theoretically grounded to achieve load balance, minimal memory-channel communication, and low space overhead. To bridge theory and practice, we incorporate implementation techniques such as practical chunking and lazy counters. Evaluation on a real-world PIM system shows that PIM-zd-tree's throughput is up to 4.25× and 99× higher than two state-of-the-art shared-memory baselines.

ACM Reference Format:
Yiwei Zhao, Hongbo Kang, Ziyang Men, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2026. PIM-zd-tree: A Fast Space-Partitioning Index Leveraging Processing-In-Memory. In *Proceedings of ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'26).* ACM, New York, NY, USA, 12 pages.

PPoPP'26, Sydney, Australia

1 Introduction

Spatial indexes for managing multi-dimensional data points are fundamental, with broad applications in computational geometry [10, 23, 37, 42, 52], AI/ML [3, 15, 38, 48, 58], graphics [13, 19, 27, 31], radars and robotics [7, 39, 40, 54, 56], and scientific simulations [9, 20, 22, 43]. Among these, some of the most well-known spatial indexes, such as kd-trees [2] and quad/octrees [44], are constructed by recursively partitioning the multidimensional space of data points—hence referred to as space-partitioning indexes. These structures support a variety of queries, including point searches, orthogonal range queries, and k-nearest neighbor (kNN) searches.

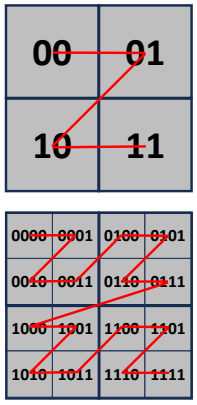


Figure 1. z-order

With the increasing amount of data in recent decades, space-partitioning indexes have become increasingly constrained by the high cost of memory access. Compared to on-chip computation, accessing off-chip memory is orders of magnitude slower and is often bottlenecked by the limited bandwidth attainable over off-chip memory channels (often referred to as the *memory wall* problem). *Processing-in-memory (PIM)*, a.k.a. *near-data-processing*, has recently gained traction as a compelling architectural paradigm to overcome the memory wall problem. By adding computational units (PIM cores) near or within memory modules, PIM enables computation to occur close to its data, in contrast to the traditional von Neumann architecture where any data must first be transferred to the CPU over off-chip memory channels. In bank-level in-memory processing (*BLIMP*) PIM designs, PIM cores are integrated directly into memory banks, enabling computation to be executed on PIM modules (PIM core and its local memory). This design improves both performance and energy efficiency by leveraging low-latency, low-energy on-chip memory accesses, while also

exploiting memory bandwidth and computational resources that scale with the number of PIM modules. Consequently, off-chip communication can be substantially reduced.

Prior work on designing space-partitioning indexes for PIM (e.g., [8, 29, 51, 57]) has been limited. This work can be broken into two groups. The first group [8, 29, 51] are (*i*) evaluated only on simulators, not real-world PIM systems, and (*ii*) lacking theoretical guarantees, which limits their ability to capture the fundamental characteristics of PIM systems. The second group [57] focuses exclusively on asymptotic theoretical analysis, with constant-factor costs and amortization overheads that will likely render such designs inefficient in practice (see §2.2 for discussion).

In this paper, the central question we aim to address is: How can a space-partitioning index be efficiently implemented on real-world PIM systems? At the same time, given the considerable diversity across existing PIM architectures, we also seek to answer the question: Can such an index be designed in a theoretically grounded manner that captures the fundamental characteristics of PIM architectures, so that it may remain effective for future PIM systems?

To this end, we set out to implement a state-of-the-art shared-memory space-partitioning index with strong theoretical guarantees on PIM systems. There are two main variants: the *k*d-trees [2] that are usually based on object-median partitioning, and quad/octree that are based on spatial-median partitioning [44]. Their state-of-the-art shared-memory parallel designs are Pkd-trees [34] and zd-trees [4, 33], respectively. Specifically, zd-trees are built by space-filling curves (Morton curve in Fig. 1).

In this work, we consider adapting zd-trees on PIM, and present **PIM-zd-tree**, the first space-partitioning index deployed (and shown to be efficient) on a *real-world* PIM system. We select the zd-tree for a few reasons. First, zd-trees are based on spatial-median partitioning, which is simpler in practice and requires no rebalancing; both are critical in achieving practical efficiency in PIM systems. Zd-trees are also deterministic, in that the structure is independent of the order of data point insertions (a.k.a. *history-independent*). This determinism simplifies programming and debugging both for developing and using the index.

Achieving an efficient PIM-based spatial index requires addressing two fundamental challenges:
(**Q1**) How can we achieve a good **trade-off** between PIM load balance, reduced off-chip communication, and low space consumption? Because PIM systems commonly operate in bulk-synchronous parallel (BSP) rounds [53], it is critical to avoid stragglers, which dictate round completion time. Achieving such balance under high workload skew, however, is particularly difficult. It often necessitates either (*i*) partitioning tasks and data at extremely fine granularity, which increases off-chip traffic, or (*ii*) replicating data across multiple PIM modules, which incurs additional space overhead and update costs.
(**Q2**) How can we efficiently bridge theoretical designs and practice? Asymptotic analyses often overlook constant factors and amortization overheads, which can introduce significant inefficiencies in real-world deployments. Hence, we require implementation techniques that both preserve theoretical bounds and deliver high performance in practice.

To address (Q1), PIM-zd-tree divides the tree into three layers based on the properties of nodes, where each layer has its own strategy for data partitioning, placement and lightweight sharing (caching). These strategies reduce off-chip communication while guaranteeing low update and space overheads. Furthermore, PIM-zd-tree is designed to be user-tunable, allowing it to support different levels of skew tolerance and varying communication and space requirements by adjusting the layer division and data management strategy correspondingly. To address (Q2) and bridge the gap between theory and practice, PIM-zd-tree incorporates a set of implementation techniques (§6) that translate theoretical insights in §5 into practical efficiency (§7).

We implement PIM-zd-tree on UPMEM [41], a real-world PIM system based on the BLIMP design. PIM-zd-tree achieves up to 4.25× and 99× speedup over Pkd-tree and zd-tree, two state-of-the-art shared-memory baselines [4, 34] and reduces memory-channel traffic by an average of 3.5× and 18.8×.

In summary, the main contributions of this paper are:

- We design PIM-zd-tree, a tunable PIM spatial index that adapts to varying requirements in skew tolerance, communication, and space overheads, with strong theoretical grounding.
- We adopt implementation techniques that effectively translate theoretical efficiency into practical performance, leveraging fundamental characteristics of BLIMP.
- We present the first implementation and evaluation of a space-partitioning index on a real-world PIM system, demonstrating significant performance gains of emerging PIM systems over traditional systems.

2 Background

2.1 PIM Architecture and Computation Model

PIM Model. In this paper, we use the Processing-In-Memory (PIM) Model [24] for theoretical analysis. Experimental results from prior work [25, 26] show that the PIM Model is a good representation of a bank-level-in-memory-processing (BLIMP) system, which is commonly used in commercial real-world PIM systems like UPMEM [41] and Samsung PIMs [47].

The PIM Model consists of a host CPU and a PIM side of P PIM modules. The CPU features a standard multicore architecture with an L3 cache sized M words. Each *PIM module* integrates a small on-chip *local memory* or *PIM memory* of $O(N/P)$ words (where N denotes total space), and a general-purpose but relatively weak processor known as the *PIM core*. Host CPU can access both its cache and the local memory of all PIM modules. However, each PIM core can only

access its own local memory. PIM modules cannot communicate directly and must exchange data via the CPU. Programs execute in bulk-synchronous parallel (BSP) rounds [53].

The PIM Model integrates both shared-memory and distributed metrics. For CPU computations, it quantifies the **CPU work** (the total number of instructions executed by the CPU) and **CPU span** (the critical path length) under a binary forking model [1, 5, 6]. For off-chip communication, it measures **communication amount** which is the sum total of words sent between CPU and all PIM modules. For PIM programs, it measures the **PIM time**, the *maximum* work on any PIM core within a round. Because PIM time is based on the maximum across all PIM modules, it is crucial to design algorithms that ensure good load balance across PIM modules, even under highly skewed query workloads.

A Real-World System: UPMEM. We evaluate our techniques on the latest PIM-based UPMEM [41] (recently acquired by Qualcomm). Its PIM are plug-and-play DRAM DIMM replacements, and therefore can be configured with various ratios of traditional DRAM memory to PIM-equipped ones (the current maximum available configuration has 2560 PIM modules). The CPU has access to both the traditional DRAM and all the PIM memory, but each PIM processor only has access to its local memory. Each PIM module has up to 628 MB/s local DRAM bandwidth, so a machine with 2560 PIM modules can provide up to 1.6 TB/s aggregate bandwidth [17]. To move data *between* PIM modules, the CPU reads from the origin and writes to the target.

UPMEM's main memory (traditional DRAM) enables running programs that overflows cache size, but these additional memory accesses bring another type of communication not existing in the PIM model: *CPU-DRAM communication*. Thus the cache efficiency is important for host programs.

2.2 Prior Works: PIM-Friendly Indexes

Space-Partitioning Indexes on PIM. Most prior works [8, 29, 51] are evaluated only on simulators, not real-world PIM systems, and lack theoretical foundations. The only work with theoretical guarantees [57] relies on periodic reconstruction of imbalanced subtrees as a core approach. However, this approach is fundamentally impractical on real systems, as its additional round complexity incurs substantial costs from *mux switch overheads* [28] in current PIM architectures.

Range-Partitioning Indexes. Early PIM-based indexes [11, 12, 32] adopt range-partitioning approaches, where the key space is divided into disjoint ranges, each stored on a PIM module. While such designs are effective in reducing communication, they are highly vulnerable to adversarial skew.

Skew-Resistant Indexes. More recent skew-resistant indexes [24–26, 57] employ finer-grained data placement and replication strategies to mitigate skew. However, most of these designs are purely theoretical [24, 26, 57] and lack practical validation. PIM-tree [25] represents an implementation effort, but (*i*) its design cannot be directly extended to spatial indexes due to fundamental structural differences, and (*ii*) it sacrifices performance in regular workloads in order to guard against skew in highly adversarial settings.

2.3 zd-Tree

The zd-Tree [4, 33], the primary data structure in our work, is a space-partitioning index of n multi-dimensional points. In short, it is a kd-tree whose splitting rule uses z-order (see Fig.1). The tree is built by letting the root represent the entire bounding box of the dataset, and splitting the points into child nodes at level i based on whether the bit at place i of the z-order key is 0 or 1. Both internal nodes and leaf nodes store information about their bounding box. Internal nodes also store their children, while leaf nodes store the point sets they contain. The number of points in a leaf is bounded by a constant, and every point is included in exactly one leaf.

From one perspective, the zd-tree is similar to an oct-tree (in 3 dimensions), except that every three levels of a zd-tree corresponds to one level of an oct-tree. From another perspective, the zd-tree can be viewed as a radix tree (or trie) whose stored keys are the z-ordered integer of the points. We adopt an implementation of a *compressed* radix tree, where we 1) omit all empty leaves, and 2) merge each node that has only one child with that child (i.e., compress all paths consisting of nodes with only one child). Note that after this compression, all internal nodes have exactly two children, and there are $2n + O(1)$ nodes in total in the zd-tree.

The zd-tree supports correct operations on arbitrary multi-dimensional datasets and has demonstrated practical efficiency [4, 33] on numerous real-world datasets. It can also achieve theoretical bounds, as detailed in the Appendix in the supplementary materials.

3 PIM-zd-tree

In this section, we introduce PIM-zd-tree—a batch-dynamic zd-tree data structure designed for Processing-In-Memory (PIM). PIM-zd-tree maintains a binary zd-tree and this section introduces the main techniques for data partitioning and replication used in our design.

A straightforward design to place a zd-tree on P PIM modules is to partition the tree into P disjoint subtrees with equal sizes, and place each tree on a different PIM module. However, such design is highly sensitive to skew in workloads—an adversary batch can target all its operations onto one PIM module and leave all the others idle.

To address this challenge, our PIM-zd-tree initially distributes each tree node across PIM modules using a hash-based randomization strategy, ensuring that adversarial operations cannot consistently target the same node. We refer to these distributed nodes as *master nodes*. However, relying solely on master nodes does not effectively reduce off-chip

communication: during searches, every tree edge incurs a remote access because parent and child nodes are typically placed on different PIM modules. As a result, the communication cost remains comparable to that of shared-memory systems, undermining the motivation for adopting PIM.

In this section, we will introduce our main design to *reduce communication* over this prototype of master node design, without violating load balance or incurring large space cost.

3.1 Overall Structure

The PIM-zd-tree divides the data structure into three layers. As shown in Fig. 2, from top (root) to bottom (leaf nodes), the tree is divided into (*i*) Level 0 (L0): globally shared nodes; (*ii*) Level 1 (L1): partially shared nodes; and (*iii*) Level 2 (L2): exclusive nodes. In each layer, a different strategy is adopted for data partitioning and caching.

Our key observation in a tree data structure is that the internal tree nodes that lie in the upper part of a tree are more frequently accessed in (top-down) searches and less frequently modified in dynamic updates compared to the nodes in lower levels. Meanwhile, the number of such nodes (the size of the upper part) is relatively small compared with the lower levels. Thus, sharing a consistent copy of the upper part nodes across different hardware modules (CPU and/or PIM modules) would be beneficial without incurring unacceptable overheads in space and update costs. Intuitively, the higher the position of a node is inside the tree, the more times it should be replicated on different modules.

Positional Descriptor of a Tree Node. We first introduce how to divide each node into its corresponding layer based on its positional information. In this paper, we use the notion of ***subtree size***—the total number of multi-dimensional data points contained in all the descendant leaf nodes of a node, denoted as $T(N_i)$ for internal node N_i—to represent the positional information of an internal node. Unlike B-trees, zd-trees are not strictly balanced. Thus, the *height* representation of each node (as in the PIM-tree [25]) might be an imprecise indicator of the position of this node inside the zd-tree. For instance, even for a dataset with bounded expansion constant (see the Appendix in the supplementary materials), a node with $\log n$ depth from the root can either be a leaf node or a node with $\Theta(\sqrt{n})$ descendants. In contrast, the subtree size is a more accurate descriptor on the position of a node, where nodes with larger subtree sizes lie in the higher parts in the tree.

The PIM-zd-tree uses two tunable thresholds, θ_{L0} and θ_{L1} ($\theta_{L0} \geq \theta_{L1} > 0$), to control the tree structure. For all nodes N_i with $T(N_i) \geq \theta_{L0}$, they are categorized as L0 nodes. Any node with $T(N_i) < \theta_{L1}$ is categorized as an L2 node. The rest are categorized as L1 nodes.

Globally Shared Nodes (L0). For the nodes in the uppermost part of the tree, their information and the tree structure will be shared globally across *all* the PIM modules. Given such storage, the concept of master nodes is unnecessary. The size of L0 could be tuned by the designer or the user.

When the size of L0 is $O(M)$ where M is the size of the CPU cache, L0 will be maintained completely in the cache. Since all queries/batch updates on PIM modules start from the host CPU side, keeping the L0 structure in CPU cache is equivalent to sharing the structure over all PIM modules.

On the other hand, when the size of L0 is $\Omega(M)$, its structure will be replicated over all PIM modules. To keep overhead of space and updates low, the size of L0 should be bounded by selecting an appropriate θ_{L0} value.

Partially Shared Nodes (L1). Each tree node in L1 will have their master node stored on a random PIM module. Tree structure of other nodes in L1 will be shared and attached to the master storage as an auxiliary structure to reduce communication in tree traversal. However, due to the large number of L1 nodes, a full global data sharing of the entire L1 structure over all PIM modules will be too costly in terms of update cost and space overhead.

In PIM-zd-tree, for each of the L1 nodes, a copy of all its ancestors and descendants (and the corresponding tree structure) in L1 will be attached to the master storage and stored on the same PIM module. This information suffices to cover all traversal paths of a search query that is passing through this L1 node, and thus the subsequent search query could be executed locally through these cached tree structures.

Exclusive Nodes (L2). For internal and/or leaf nodes with $T(N_i) < \theta_{L1}$, due to their large total number, high frequency in updates and unlikeliness in facilitating search queries, a PIM-zd-tree does not make replicas of these nodes and only stores their master copies. Each leaf node is allowed to hold a maximum of B data points (defined later in §3.2).

Promotion and Demotion. PIM-zd-tree assumes that structural changes to the tree occur only through dynamic updates (insertions and deletions). Such updates affect the subtree sizes of all internal nodes along the path from the root to the modified leaf node. Roughly, if the updated size of an internal node causes its transition between categories (L0/L1/L2), the node is *promoted* or *demoted* accordingly, and its associated caching for data sharing is adjusted to reflect the change.

3.2 Chunking for Imbalanced Trees

Chunking (or blocking) is a widely adopted technique in locality-aware data structures, such as transforming binary search trees into B-trees. However, traditional chunking methods that rely on adjusting fanout (as in B-trees) are incompatible with imbalanced trees like zd-trees, since the fanout approach fundamentally assumes that tree height and level are well-defined and meaningful.

PIM-zd-tree employs a chunking strategy for its L1 and L2 layers that is based entirely on subtree sizes, guided by a user-defined chunking factor B. For a highest node N_i in L1/L2, all descendants N_j satisfying $T(N_j) \geq T(N_i)/B$

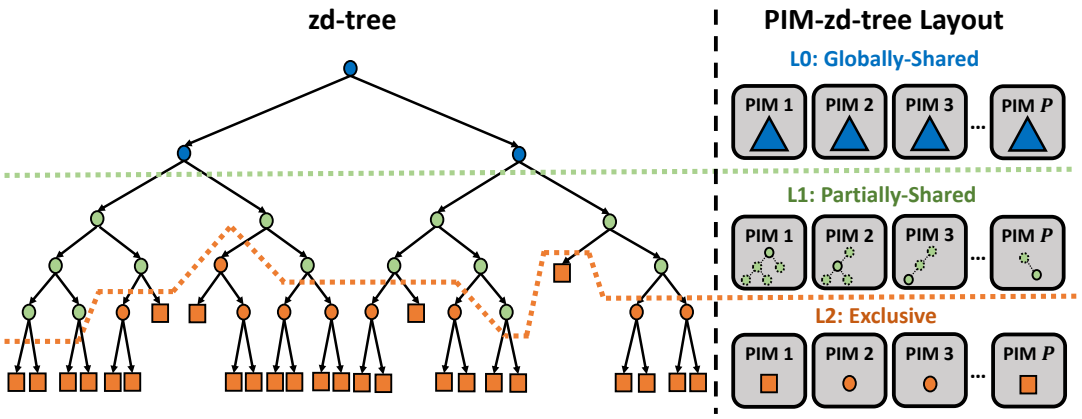


Figure 2. PIM-zd-tree with its three-layer structure and corresponding data layout across PIM modules. Squares denote leaves, while circles represent internal nodes. Solid-frame nodes indicate master storage, whereas dashed-frame nodes represent cached data used in data sharing.

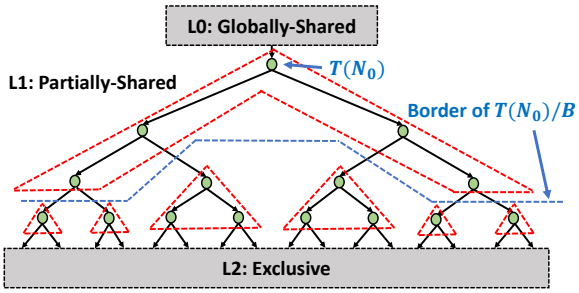


Figure 3. Example of dividing an L1 subtree into meta-nodes. Each region enclosed by a red dashed line corresponds to a meta-node. The L1 root has subtree size $T(N_0)$, while the blue dashed line marks the boundary below which all nodes satisfy $T(N_i) < T(N_0)/B$. Recursive division would be applied if any L1 nodes further satisfy $T(N_i) < T(N_0)/B^2$, but this step is omitted for clarity.

are grouped into the same chunk as N_i. Each chunk, which naturally forms an imbalanced tree, is referred to as a ***meta-node***. This chunking process is then applied recursively to the highest unchunked nodes until every node is contained within some meta-node. An example is shown in Fig. 3.

All nodes within the same meta-node are placed on a single PIM module, and any internal caching for data sharing among them is now eliminated. The original zd-tree is thus restructured into a higher-level tree composed of meta-nodes. For nodes in L1, remote caching to enable data sharing across different meta-nodes is preserved, but maintained at the granularity of meta-nodes rather than individual nodes.

3.3 Push-Pull Search

We employ *push-pull search* [25, 26] as a core technique in PIM-zd-tree to achieve load balance when accessing L1 and L2 nodes under skewed workloads. Unlike distributed systems, PIM architectures feature a powerful host processor, enabling the use of shared-memory techniques. Push-pull search exploits this capability by flexibly coordinating computation between the host and PIM modules, in contrast to prior distributed algorithms that rely solely on offloading.

The push-pull search uses contention information within a batch to decide whether computation in a PIM-zd-tree is performed on the CPU side or on PIM modules. To illustrate, consider a top-down Search query from root to leaf. In a PIM execution round with batch size S, suppose m queries need to access an L1 meta-node M_i to determine which descendant subtree to follow. If m is below a load-imbalance threshold (defined later), all m queries are *pushed* to the PIM module storing M_i, where the search proceeds using the local caching of its descendant subtrees until it reaches L1-L2 border. Conversely, if m exceeds the threshold, then M_i will be *pulled* from the PIM side into the CPU, where a parallel search is performed. Notably, only the master storage of the meta-node is fetched; caching of its descendant meta-nodes is excluded to prevent communication imbalance. After the CPU search, the m queries are partitioned according to the destination descendants from their meta-node traversal, and the push-pull decision is recursively applied level by level at the granularity of meta-nodes until the leaves are reached.

3.4 Lazy Counters

As noted, subtree size is a key component in the design of the PIM-zd-tree. However, maintaining an accurate and consistent version of this metric across all nodes is challenging. A straightforward approach would involve storing precise counters on every node and its replicas, while ensuring consistency during dynamic updates. Yet, the overall strategy of the PIM-zd-tree is to replicate higher-level nodes more extensively. As a result, changes in subtree counters propagate toward the upper levels of the tree, making it prohibitively expensive to maintain strict consistency during updates.

A key observation is that a provably small degree of approximation is acceptable in these counters. Prior work has explored the design of randomized counters [35, 49, 50, 55, 57]. However, employing randomization in practice on PIM systems can lead to irregular and unpredictable execution flows, in addition to the overhead of random number generation or hash computations on lightweight PIM cores.

To resolve this challenge, PIM-zd-tree adopts ***lazy counters***, which in each node maintains a slightly out-of-date global snapshot of the subtree sizes. This snapshot is within a degree of approximation to ensure algorithms' correctness, is replicated across caching, and is infrequently updated.

In summary, at lower levels, nodes records changes in their subtree sizes during dynamic updates. Changes are propagated to parent nodes and global snapshots are synchronized across all replicas on other PIM modules only when the change exceeds $\epsilon \cdot T(N_i)$, where ϵ is a small positive constant. These lazy counters achieve sufficient accuracy, as formalized in Lemma 3.1. Due to space constraints, readers interested in the detailed description and theoretical analysis are referred to the Appendix in the supplementary materials.

Lemma 3.1 (Lazy Counter Value)**.** *For cases with both insertions and deletions, the value of global snapshot SC_i always satisfies $T(N_i)/2 \leq SC_i \leq 2T(N_i)$.*

4 Operations

We describe here our algorithms for implementing PIM-zd-tree and various queries over them. The theoretical analysis on the cost of these algorithms will be provided in §5.

4.1 Top-Down Search

Top-down Search queries try to locate the leaf node where a given data point lies in. This can be used as a preprocessing for dynamic updates and kNN queries.

Algorithm 1. Search (Q: batch of query points)

1. [$L0$] Traverse L0 to search Q by (1) searching in CPU cache; or (2) dividing Q into P groups, each searched on a PIM module.
2. [$L1$ Pull] While the number of queries that will be sent to each PIM module for L1 is imbalanced (i.e., the busiest module gets more than 3× the average load), do:
   a. Pull all meta-nodes with more than $K = B \log_B \frac{\theta_{L0}}{\theta_{L1}}$ queries back to the CPU.
   b. Search through the meta-nodes on CPU.
3. [$L1$ Push] Push load-balanced queries in Q to the PIM modules holding their L1 nodes, and traverse L1 using local caching.
4. [$L2$ Push-Pull] Perform multiple push-pull rounds with $K = B$ to traverse L2, and retrieve the data points.

4.2 Dynamic Updates

Insert adds new data points to PIM-zd-tree, while Delete removes existing data points from the structure. Due to the complexity of the algorithm, we will provide the details of this algorithm in the full version on arXiv.

4.3 k Nearest Neighbors

A k nearest neighbor (kNN) query requires to return the exact k nearest points in PIM-zd-tree to a given point with a pre-defined distance metric (e.g., l1-norm or l2-norm).

Algorithm 2. KNN (Q: batch of query points; k: INT)

1. Search (Q) and record the search traces.
2. For each $q \in Q$, find on the search trace the lowest node whose lazy counter records an subtree size of $\geq k$. Use push-pull search to traverse its descendants to find k nearest candidate neighbors.
3. Find on the search trace the lowest node which entirely contains the bounding sphere formed up by the k candidates.
4. Use push-pull search to traverse its descendants who intersect with the bounding sphere of the k candidates. Return the points which lie in the sphere.
5. Filter out the final kNN results on CPU.

4.4 Orthogonal Range Query

An orthogonal range query, or box query, specifies one (or a batch of) axis-aligned rectangular boxes. There are two categories based on their output: a BoxCount query returns the number of points in PIM-zd-tree that fall within the box, while a BoxFetch query retrieves all such points. The execution procedure closely follows that of Search, where push-pull search is applied level by level. The key difference is that box queries must also track all nodes intersecting the query box. Pseudocode is omitted due to space constraints.

5 Theoretical Analysis

In this section, we analyze PIM-zd-tree and show that it achieves good performance regardless of skew in the PIM model. Our analysis in this section assumes bounded ratio and bounded expansion constant (which can be found in the Appendix in the supplementary materials). However, this does not mean that PIM-zd-tree is only practically efficient under such dataset distributions. In §7, we will show that the design choices we made in PIM-zd-tree are practically efficient in various real-world datasets, regardless of whether these datasets satisfy the two assumptions or not. For readers less interested in the mathematical details, we suggest focusing on the main conclusions of Theorems 5.1 and 5.3 to 5.5; or directly on Table 1, which summarizes the configurations implemented on real-world machines.

Before analyzing the theoretical costs of different operations, we first define (α, β)-skew in Definition 1 to asymptotically characterize the skew in a batch distribution.

Definition 1 (Skew). *A batch of S queries with keys in the range $[U_l, U_r]$ is defined to have (α, β)-skew iff for every integer $\forall i \in [1, \beta]$, the number of keys falling in the subinterval $\left[U_l + \frac{i-1}{\beta}(U_r - U_l), U_l + \frac{i}{\beta}(U_r - U_l)\right]$ is at most $\frac{S}{\alpha}$.*

5.1 Space Consumption

Theorem 5.1 (Space). *A PIM-zd-tree containing n data points takes $O\left(n + \frac{nP}{\theta_{L0}} + \frac{n}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}}\right)$ space.*

Proof Sketch. Due to bounded expansion ratio, L0 has $O(n/\theta_{L0})$ nodes. L1 has $O(n/\theta_{L1})$ nodes, each of which is replicated $O\left(\log_B \frac{\theta_{L0}}{\theta_{L1}}\right)$ times (the maximum height of an L1 path). □

5.2 Top-Down Search

The costs for Search is presented in Theorem 5.3. By properly tuning θ_{L0}, θ_{L1} and B, the PIM-zd-tree can be adapted to varying degrees of (α, β)-skew, ensuring a desired cost bound. Load balance is proved using Lemma 5.2.

Lemma 5.2 (Balls into Bins [45]). *Uniformly randomly placing weighted balls with total weight $W = \sum w_i$ and $w_i < W/(P \log P)$ into P bins yields $O(W/P)$ weight per bin whp*[1].

Theorem 5.3 (Search). *A batch of S Search queries can be executed in worst-case $O(\log_B \theta_{L0})$ communication rounds, and takes a total of $O(S \log_B \theta_{L1})$ communication amount and $O(S \log n)$ PIM work. The PIM execution is load-balanced whp if the batch size is $S = \Omega(P \log P \cdot B \log_B \theta_{L0})$ or if the batch has $\left(P \log P \log_B \theta_{L0}, \frac{n}{\theta_{L0}}\right)$-skew. On CPU it takes $O(S \log_B \theta_{L1})$ expected work and $O(\log S \log_B \theta_{L1} + \log_B \theta_{L0})$ span whp.*

Proof Sketch. The PIM work is $O(S \log_B n)$ to search the S queries through the tree. The worst-case communication round is the total height of L1 and L2 (due to pulling at every level), which is $O(\log_B \theta_{L0})$. Communication amount for each Search query, due to the amortization of push-pull search, is $O(1)$ for L0 and L1, and is height $O(\log_B \theta_{L1})$ for L2. When either the batch size or the skew condition holds, the balance in PIM computation and communication can be proved using Lemma 5.2. The proof for CPU execution uses parallel work-efficient semi-sort [16] and radix sort [21]. □

5.3 Dynamic Updates

In this section, we present the overall cost of dynamic updates. We restrict our analysis here to the insertion-only case due to space constraints.

Theorem 5.4 (Insert). *A batch of S Insert can be executed in worst-case $O(\log_B \theta_{L0})$ communication rounds, and takes a total of $O\left(\frac{SP}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} + S \log_B \theta_{L1}\right)$ communication amount and $O\left(\frac{SP}{\theta_{L0}} \log \frac{n}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} \log \frac{\theta_{L0}}{\theta_{L1}} + S \log n\right)$ PIM work. The PIM execution is load-balanced whp if batch size $S = \Omega\left(P \log P \cdot \left(B \log_B \theta_{L0} + \log \frac{\theta_{L0}}{\theta_{L1}}\right)\right)$ or if the batch has $\left(P \log P \log_B \theta_{L0}, \frac{n}{\theta_{L0}}\right)$-skew and $S = \Omega(P \log P \log \frac{\theta_{L0}}{\theta_{L1}})$. CPU execution takes $O\left(\frac{SP}{\theta_{L0}} + \frac{S}{\theta_{L1}} \log_B \frac{\theta_{L0}}{\theta_{L1}} + S \log_B \theta_{L1}\right)$ expected work and $O(sort(S) + \log S \log_B \theta_{L1} + \log_B \theta_{L0})$ span whp.*

Proof Sketch. For the proof of computation work, we refer to the proof of shared-memory zd-trees [4]. For the communication analysis, we combine Theorem 5.3 and the lazy counter update cost in the Appendix of the supplementary materials, and additional costs in updating the data sharing structures. The update costs of data sharing structures are in the same frequency as lazy counters: updating P L0 copies in amortized every $\Theta(\theta_{L0})$ updates in expectation, and $\Theta(\log_B \frac{\theta_{L0}}{\theta_{L1}})$ L1 copies every $\Theta(\theta_{L1})$ updates in expectation, so a similar analysis in the Appendix can be used here. □

5.4 k Nearest Neighbors

The costs for kNN are provided in Theorem 5.5. Please refer to the Appendix in supplementary materials for its proof.

Theorem 5.5 (kNN). *Finding the k nearest neighbors of a data point requires expected $O(k + \log_B \theta_{L1})$ communication amount, worst-case $O(\log_B \theta_{L0})$ communication rounds, expected $O(k + \log n)$ PIM work, expected $O(k \log k + \log_B \theta_{L1})$ CPU work and expected $O(k)$ CPU cache footprint.*

Table 1. Configurations of implementations, space consumption, and communication amount per operation.

Method	Throughput-Optimized	Skew-Resistant
θ_{L0}	n/P	$\Theta(P)$
θ_{L1}	1	$\Theta(\log_B P)$
B	θ_{L0}	$\Theta(1) = 16$
Allowed Skew	$(P \log P, \frac{n}{P})$	Arbitrary
Required S	$\Omega(P \log P)$	$\Omega(P \log^2 P)$
Space	$O(n)$	$O(n)$
Search	$O(1)$	$O(\log_B \log_B P)$
Updates	$O(1)$	$O(\log_B \log_B P)$
kNN	$O(k)$	$O(k + \log_B \log_B P)$

6 Implementation

To demonstrate the practical efficiency of our methods, we implement two configurations of PIM-zd-tree that represent the two extremes of the design frontier. The first is a throughput-optimized version, which prioritizes communication and computation efficiency. The second is a skew-resistant version, capable of tolerating arbitrary adversarial skew when $S = \Omega(P \log^2 P)$. The key configurations and the operation costs are summarized in Table 1.

We implement our methods on UPMEM [17, 41]. In this section, we describe the practical techniques adopted to

[1] We use $O(f(n))$ **with high probability** (***whp***) (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

achieve high performance. Most of these techniques build on fundamental characteristics of BLIMP, and we expect them to apply to a wide range of architectures beyond UPMEM.

Practical Chunking. We provide an adaptive node structure design for L1 with two capacity modes, which is inspired by the adaptive radix tree (ART) [30]. It implements the imbalanced-tree-shaped chunking described in §3.2. The key observation is that dense internal nodes are highly likely to appear in the inner levels of the tree, where the chunk structure tends to be well balanced. In contrast, sparse internal nodes are more likely to occur near the leaf levels, where the chunk structure may become imbalanced.

- **Sparse Mode**: If an L1 chunk contains fewer than $B/4$ nodes, we use two arrays of length $B/4$—one for keys and one for pointers. Keys are stored in sorted order, and each pointer is aligned with its corresponding key.
- **Dense Mode**: If the chunk contains at least $B/4$ nodes, we instead use an array of B pointers to represent the root node of the chunk. A descendant can be located with a single lookup using the key byte as an index.

Fast z-Order Computation. Computing the z-order of the keys is a critical step in the performance of PIM-zd-tree. Most prior works adopt the direct bit-wise interleaving method (e.g., [4, 33]), which has a complexity of $O(\text{bits})$. In contrast, our implementation employs a faster z-order computation based on the recursive construction of gaps over the original coordinates, reducing the complexity to $O(\log(\text{bits}))$. As an example, we illustrate below how 3D data points are transformed into 64-bit keys:

```
function Split_By_Three(uint64 x) { // x in [0, 2^21]
    x = (x | (x << 32)) & 0x001f00000000ffff;
    x = (x | (x << 16)) & 0x001f0000ff0000ff;
    x = (x | (x << 8))  & 0x100f00f00f00f00f;
    x = (x | (x << 4))  & 0x10c30c30c30c30c3;
    x = (x | (x << 2))  & 0x1249249249249249;
    return x;
}
function Z_Order_Key_3d(uint64 x, y, z) {
    x = Split_By_Three(x);
    y = Split_By_Three(y);
    z = Split_By_Three(z);
    return (x << 3) | (y << 2) | (z << 1);
}
```

Execution of Complex Distance Metrics on PIMs. Due to limited areas inside memory chips, BLIMP architectures often suffer from constrained computational power on the PIM cores. This makes the computation of complex distance metrics on the PIM side comparatively slow. For example, on UPMEM machines, multiplication and division may take up to 32 cycles, much slower than simpler arithmetic operations (e.g., addition or bitwise AND/OR) [17], which significantly hinders the efficient computation of the ℓ_2-norm on PIM.

We propose an efficient execution flow for cases where a complex distance metric can be *anchored* by a simpler one. For example, the ℓ_2-norm can be anchored by the ℓ_1-norm since, for any $x \in \mathbb{R}^D$, $||x||_2/||x||_1 \in [1/\sqrt{D}, 1]$. As a result, in a kNN query, if the k-th nearest neighbor under ℓ_1-norm has distance x, then the k-th nearest neighbor under ℓ_2-norm must have ℓ_1-distance of at most $x\sqrt{D}$.

We decompose the candidate-finding process in Algorithm 2 into two stages: coarse-grained filtering and fine-grained filtering. In the coarse-grained stage, PIM cores use a simple-to-compute (on PIM) distance metric (e.g., ℓ_1-norm) to filter out a small candidate set that is guaranteed to contain all k nearest neighbors. The fine-grained stage is executed on the CPU, where a more complex distance metric (e.g., ℓ_2-norm) produces the accurate final results. For low-dimensional $D = O(1)$ and the ℓ_1- and ℓ_2-norm case, the candidate set returned by the coarse-grained filtering still has size $O(k)$, and the bounds in Algorithm 2 remain valid.

Improved Direct API. We implement a lightweight *Direct Interface/API* to mitigate the overhead of the original UPMEM interface in small-batch scenarios. Further details are provided in the Appendix of the supplementary materials.

7 Evaluation

7.1 Experimental Setup

We evaluate PIM-zd-tree on an UPMEM®PIM-equipped server. The server features two Intel Xeon Silver 4216 CPUs (32 threads total, 2.1 GHz, 22 MB LLC) and 12 memory channels, of which eight are populated with UPMEM DIMMs and four with standard DDR4 2400MT/s DRAM DIMMs. In total, the system includes 32 UPMEM ranks (2048 modules) providing 128 GB of PIM memory, with PIM cores running at 350 MHz.

Shared-Memory Competitors. We compare the throughput-optimized PIM-zd-tree against two state-of-the-art shared-memory implementations–zd-tree [4] and Pkd-tree [34].

We cannot evaluate shared-memory indexes on the UPMEM server because two-thirds of its memory channels are occupied by PIM-equipped DIMMs, which cannot serve as main memory. Running shared-memory indexes directly on this server would therefore create an unfair limitation on memory bandwidth. Instead, we evaluate shared-memory indexes on a separate machine equipped with two Intel Xeon E5-2630 v4 CPUs, each with 10 cores at 2.20GHz and 25MB cache. Each socket has four memory channels, and no PIM-equipped DIMMs are present. This machine is similar but slightly *more powerful* than the UPMEM server, as we could not find an exact match.

Measurement. We evaluate on two metrics: (*i*) *Throughput*: Defined as the number of returned elements per second. For point operations (Insert, BoxCount), throughput corresponds to the number of operations executed per second, whereas for range operations (BoxFetch, kNN), it represents

the number of elements returned in the final output per second. (*ii*) *Per-Element Memory Traffic*: Defined as the total memory-bus communication (in bytes) incurred per returned element in the final output, including both DRAM-CPU and PIM-CPU communication. Memory traffic is a primary contributor to power consumption in index-based applications (see [17, 25, 36] for detailed studies on energy consumption).

7.2 End-to-End Comparison

Workload Setup. We begin with a microbenchmark using a uniformly random dataset. Each test first warms up the index by inserting 300 million uniformly random 3D data points. The benchmark then executes (*i*) 50 million point operations or (ii) range operations that retrieve a total of 50 million elements in expectation. For BoxCount, BoxFetch and *k*NN queries, we evaluate each with three different query range sizes, covering on average 1, 10, and 100 data points.

In addition, we evaluate on two real-world datasets: COSMOS (CM) [46] and the Northern American region of OpenStreetMap (OSM) [18]. For each dataset, we use 80% of the data points for warmup and the remaining 20% for testing.

Main Results. Fig. 4 presents the main results comparing the throughput-optimized PIM-zd-tree with Pkd-tree and zd-tree across ten types of operations. PIM-zd-tree achieves geometrically averaged speedups of 1.82×, 4.25×, 3.08×, and 1.46× over Pkd-tree for Insert, BoxCount, BoxFetch, and *k*NN, respectively. Against zd-tree, the corresponding speedups are 1.49×, 518×, 99×, and 3.46×. The geometrically averaged memory traffic reduction across all operations is 3.5× compared to Pkd-tree and 18.8× compared to zd-tree.

The few cases where PIM-zd-tree does not outperform Pkd-tree in throughput occur for *k*NN queries with large *k* values. This is because large *k*NN queries are more likely to cross the boundaries between PIM modules, resulting in multiple rounds of communication and incurring significant *mux switch overhead* [28] when switching control of PIM memory between CPU and PIM-core accesses.

7.3 Ablation Study

Breakdown of Time. Fig. 5 illustrates the time breakdown of CPU computation, PIM computation, and CPU–PIM communication. The Insert operation incurs significant CPU time, primarily due to preprocessing over the batch. In contrast, BoxFetch with size 100 exhibits high CPU–PIM communication time, as its computation is simple but the output size is large. For all other operations, the majority of the time is spent on PIM execution, which aligns with our design goal of offloading computation to PIM.

Effect of Batch Sizes. Batch size plays a critical role in the execution of PIM-zd-tree. Larger batch sizes are preferred to amortize the mux switch overhead [28] and to achieve effective load balance. However, excessively large batches, combined with auxiliary structures, may exceed the capacity of the L3 cache, resulting in increased memory traffic.

Fig. 6 presents an ablation study on the impact of different batch sizes for Insert operations. While increasing the batch size improves throughput, batch sizes exceeding 200k operations result in higher memory traffic per operation. This finding suggests that future systems with larger caches would be advantageous. Similar trends were observed for box and *k*NN queries, but are omitted here due to space limit.

Sensitivity to Dataset Sizes. One theoretical result we obtain is that, while search paths in shared-memory indexes have a length bounded by $O(\log n)$, the communication cost of PIM-zd-tree, as shown in §5, is bounded solely by the number of PIM modules P and is independent of n. Consequently, PIM-zd-tree is expected to maintain robust performance across datasets of varying sizes.

We evaluate the performance of the three methods under varying base dataset sizes during warmup, as shown in Fig. 7. The performance of PIM-zd-tree remains stable across dataset sizes, whereas the throughput of Pkd-tree and zd-tree degrades by 1.4× and 1.6×, respectively. Correspondingly, their memory traffic increases by 1.3× and 1.5×.

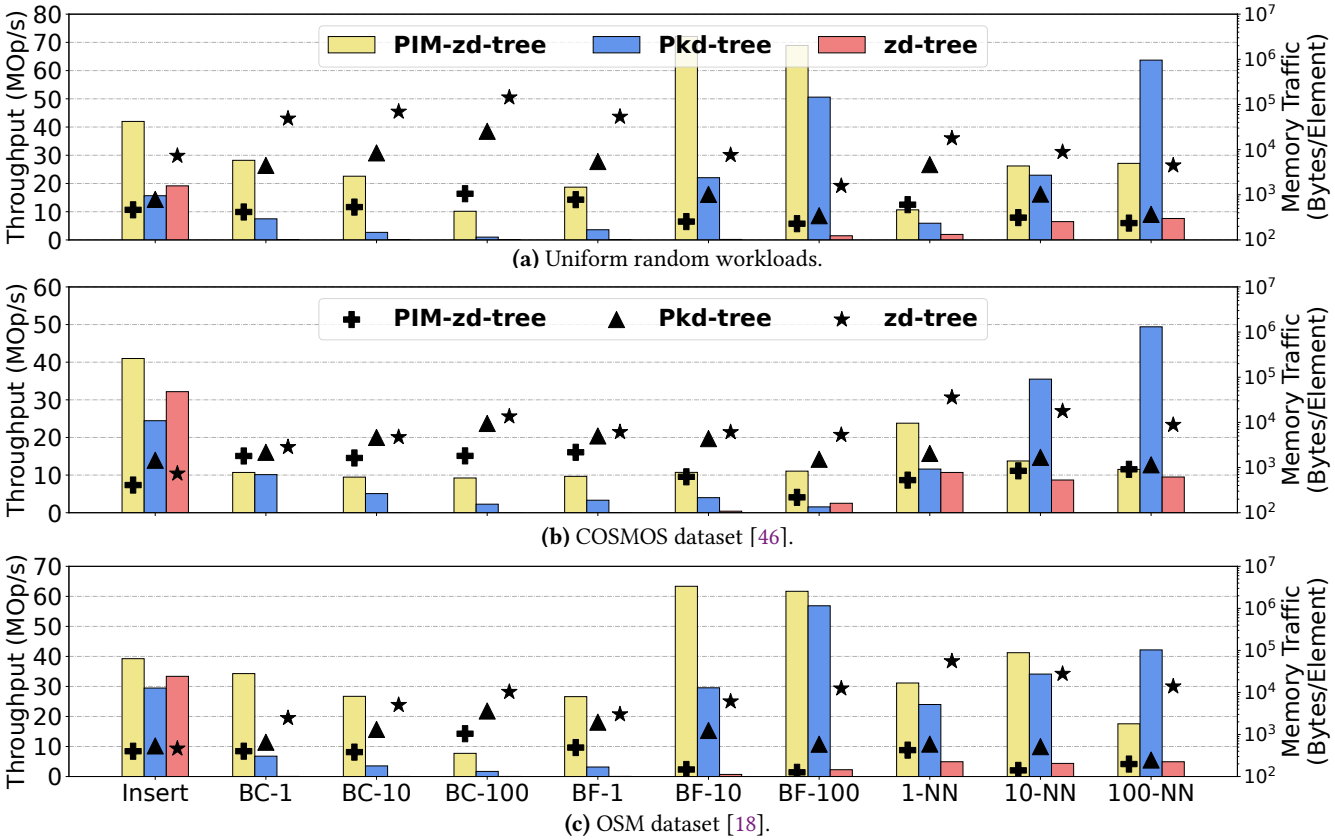
Sensitivity to Optimizations. We evaluate the impact of four implementation techniques in PIM-zd-tree: lazy counter, fast z-order, fast ℓ_2-norm, and improved Direct API. Table 2 reports the slowdown observed when each technique is individually removed from the final design. All techniques provide substantial performance benefits, with the exception of Direct API. The limited impact of Direct API arises from our use of large batch sizes to maximize performance, which falls outside the scenarios for which it is primarily optimized.

Table 2. Impact of implementation techniques on slowdown under uniform workloads. Results for box queries and *k*NN are reported as geometric means across three query sizes. *N.A.* indicates that the corresponding technique is not optimized for the given operation.

Technique	Insert	BoxCount	BoxFetch	*k*NN
Lazy Counter	1.49×	N.A.	N.A.	N.A.
Fast z-order	1.99×	1.58×	1.31×	1.67×
Fast ℓ_2-norm	N.A.	N.A.	N.A.	1.58×
Direct API	1.06×	1.07×	1.09×	1.09×

Skew Resistance. We further evaluate the skew-resilience of PIM-zd-tree under non-uniform workloads. Specifically, we compare the performance of both the throughput-optimized and the skew-resistant versions on *k*NN under skewed conditions. The skewed workload is derived from Varden [14], an extremely skewed distribution generated via random walk.

In our experiments, we mix *k*NN queries generated from the skewed Varden distribution into batches of uniformly distributed *k*NN queries. Fig. 8 presents the resulting throughput across varying proportions of skewed queries. The skew-resistant version of PIM-zd-tree demonstrates highly stable



(a) Uniform random workloads.

(b) COSMOS dataset [46].

(c) OSM dataset [18].

Figure 4. Comparison of PIM-zd-tree, zd-tree, and Pkd-tree across three datasets on Insert, BoxCount (BC), BoxFetch (BF) and nearest neighbor (NN) operations. The bar plots report throughput, while the scatter plots show memory traffic, measured as the number of bytes transmitted through the memory bus per element in the final output.

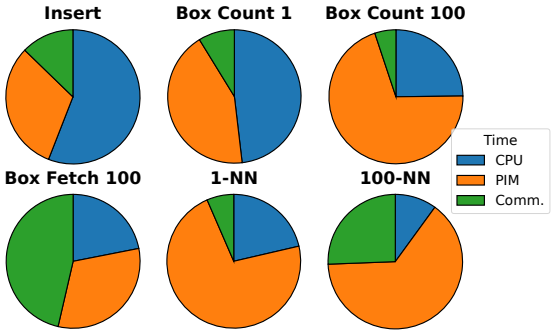


Figure 5. Runtime breakdown of different operations.

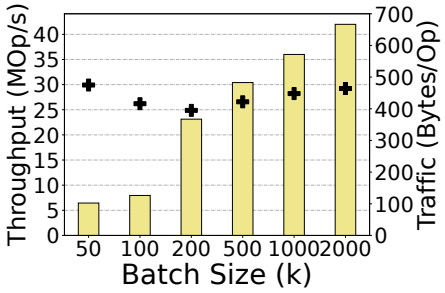


Figure 6. Insert performance given different batch sizes.

performance, with fluctuations of no more than 4.1%. In contrast, while the throughput-optimized version performs exceptionally well under workloads with low degrees of skew, it is outperformed by the skew-resistant variant when more than 0.1% of the workload is skewed, and its performance degrades by 10.66× when 2% of the queries originate from the Varden distribution.

8 Conclusion

We present PIM-zd-tree, the first space-partitioning index evaluated on a real-world PIM system. Our design introduces a tunable structure that adapts to different requirements in skew tolerance, communication, and space overheads. We further adopt implementation techniques that ensures practical performance. PIM-zd-tree delivers up to 4.25× and 518× speedup over two shared-memory baselines.

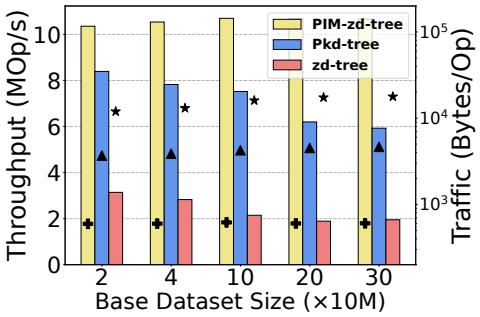


Figure 7. 1-NN throughput and memory traffic given different base dataset sizes during warmup before testing.

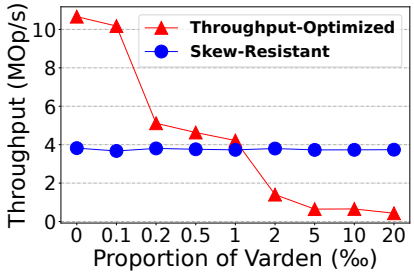


Figure 8. 1-NN throughputs of thoughput-optimized and skew-resistant PIM-zd-tree, given combinations of Uniform+Varden [14].

References

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

[2] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975).

[3] Wenhao Bi, Junwen Ma, Xudong Zhu, Weixiang Wang, and An Zhang. 2022. Cloud service selection based on weighted KD tree nearest neighbor search. *Applied Soft Computing* 131 (2022), 109780. doi:10.1016/j.asoc.2022.109780

[4] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[5] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2020), 89–102.

[6] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[7] Yixi Cai, Wei Xu, and Fu Zhang. 2021. ikd-Tree: An Incremental K-D Tree for Robotic Applications. arXiv:2102.10808 [cs.RO] https://arxiv.org/abs/2102.10808

[8] Mingkai Chen, Cheng Liu, Shengwen Liang, Lei He, Ying Wang, Lei Zhang, Huawei Li, and Xiaowei Li. 2024. An Energy-Efficient In-Memory Accelerator for Graph Construction and Updating. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 6 (2024), 1781–1793. doi:10.1109/TCAD.2024.3355038

[9] Qile P Chen, Bai Xue, and J Ilja Siepmann. 2017. Using the k-d tree data structure to accelerate Monte Carlo simulations. *Journal of Chemical Theory and Computation* 13, 4 (2017), 1556–1565.

[10] Yewang Chen, Lida Zhou, Yi Tang, Jai Puneet Singh, Nizar Bouguila, Cheng Wang, Huazhen Wang, and Jixiang Du. 2019. Fast neighbor search by using revised k-d tree. *Information Sciences* 472 (2019), 145–162. doi:10.1016/j.ins.2018.09.012

[11] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2022. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. 321–332.

[12] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 297–308.

[13] Byeongjun Choi, Byungjoon Chang, and Insung Ihm. 2013. Improving Memory Space Efficiency of Kd-tree for Real-time Ray Tracing. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 335–344.

[14] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 14 (July 2017), 45 pages. doi:10.1145/3083897

[15] Seyedeh Gol Ara Ghoreishi, Charles Boateng, Sonia Moshfeghi, Muhammad Tanveer Jan, Joshua Conniff, Kwangsoo Yang, Jinwoo Jang, Borko Furht, David Newman, Ruth Tappen, et al. 2025. Quad-tree Based Driver Classification using Deep Learning for Mild Cognitive Impairment Detection. *IEEE Access* (2025).

[16] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[17] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022), 52565–52608. doi:10.1109/ACCESS.2022.3174101

[18] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.

[19] Xu-Qiang Hu and Yu-Ping Wang. 2023. QuadSampling: A Novel Sampling Method for Remote Implicit Neural 3D Reconstruction Based on Quad-Tree. In *International Conference on Computer-Aided Design and Computer Graphics*. Springer, 314–328.

[20] Yuan Huang, Zhiqin Zhao, Conghui Qi, Zaiping Nie, and Qing Huo Liu. 2018. Fast Point-Based KD-Tree Construction Method for Hybrid High Frequency Method in Electromagnetic Scattering. *IEEE Access* 6 (2018), 38348–38355. doi:10.1109/ACCESS.2018.2853659

[21] J. JaJa. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[22] Shouyan Jiang, Liguo Sun, Ean Tat Ooi, Mohsen Ghaemian, and Chengbin Du. 2022. Automatic mesoscopic fracture modelling of concrete based on enriched SBFEM space and quad-tree mesh. *Construction and Building Materials* 350 (2022), 128890.

[23] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2017. A progressive k-d tree for approximate k-nearest neighbors. In *2017 IEEE Workshop on Data Systems for Interactive Analysis (DSIA)*. 1–5. doi:10.1109/DSIA.2017.8339084

[24] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 295–306.

[25] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proc. VLDB Endow.* 16, 4 (dec 2022), 946–958. doi:10.14778/3574245.3574275

[26] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2023. PIM-Trie: A Skew-Resistant Trie for Processing-in-Memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3558481.3591070

[27] Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang. 2013. gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *Journal of Systems Architecture* 59, 3 (2013), 166–175.

[28] Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. 2025. No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory. *Proc. VLDB Endow.* (2025), 4241–4254. doi:10.14778/3749646.3749690

[29] Vincent T. Lee, Amrita Mazumdar, Carlo C. del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. 2018. Application Codesign of Near-Data Processing for Similarity Search. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 896–907. doi:10.1109/IPDPS.2018.00099

[30] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. doi:10.1109/ICDE.2013.6544812

[31] Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1595–1598.

[32] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-memory Computing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[33] Magdalen Dobson Manohar, Yuanhao Wei, and Guy E. Blelloch. 2025. CLEANN: Lock-Free Augmented Trees for Low-Dimensional k-Nearest Neighbor Search. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, OR, USA) *(SPAA '25)*. Association for Computing Machinery, New York, NY, USA, 131–143. doi:10.1145/3694906.3743339

[34] Ziyang Men, Zheqi Shen, Yan Gu, and Yihan Sun. 2025. Parallel kd-tree with Batch Updates. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[35] Robert Morris. 1978. Counting large numbers of events in small registers. *Commun. ACM* 21, 10 (1978), 840–842.

[36] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2023. *A Modern Primer on Processing in Memory*. Springer Nature Singapore, Singapore, 171–243. doi:10.1007/978-981-16-7487-7_7

[37] Mohammed Otair. 2013. Approximate k-nearest neighbour based spatial clustering using k-d tree. arXiv:1303.1951 [cs.DB] https://arxiv.org/abs/1303.1951

[38] Gonçalo Perrolas, Milad Niknejad, Ricardo Ribeiro, and Alexandre Bernardino. 2022. Scalable fire and smoke segmentation from aerial images using convolutional neural networks and quad-tree search. *Sensors* 22, 5 (2022), 1701.

[39] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. 2020. QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 180–192. doi:10.1109/HPCA47549.2020.00024

[40] KH Vijayendra Prasad and P Sasikumar. 2024. Energy-efficient quad tree-based clustering using edge-assisted UAV-relay to enhance network lifetime in WSN. *Scientific Reports* 14, 1 (2024), 17160.

[41] Qualcomm. 2025. UPMEM Technology. https://www.upmem.com/technology/. Accessed August 2025.

[42] Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1378–1388. doi:10.1145/3292500.3330875

[43] W. Saftly, M. Baes, and P. Camps. 2014. Hierarchical octree and k-d tree grids for 3D radiative transfer simulations. *A&A* 561 (2014), A77. doi:10.1051/0004-6361/201322593

[44] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June 1984), 187–260. doi:10.1145/356924.356930

[45] Peter Sanders. 1996. On the Competitive Analysis of Randomized Static Load Balancing. In *Workshop on Randomized Parallel Algorithms (RANDOM)*.

[46] Nick Scoville, H Aussel, Marcella Brusa, Peter Capak, C Marcella Carollo, M Elvis, M Giavalisco, L Guzzo, G Hasinger, C Impey, et al. 2007. The cosmic evolution survey (COSMOS): overview. *The Astrophysical Journal Supplement Series* 172, 1 (2007), 1.

[47] Samsung Semiconductor. Accessed August 2025. HBM-PIM: Cutting-edge memory technology to accelerate next-generation AI. https://semiconductor.samsung.com/news-events/tech-blog/hbm-pim-cutting-edge-memory-technology-to-accelerate-next-generation-ai/.

[48] Yunxiao Shan, Shu Li, Fuxiang Li, Yuxin Cui, Shuai Li, Ming Zhou, and Xiang Li. 2022. A density peaks clustering algorithm with sparse search and Kd tree. *IEEE Access* 10 (2022), 74883–74901.

[49] Guy L Steele Jr and Jean-Baptiste Tristan. 2016. Adding approximate counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[50] Guy L Steele Jr and Jean-Baptiste Tristan. 2018. Method and system for latent dirichlet allocation computation using approximate counters. US Patent 10,147,044.

[51] Boyu Tian, Qihang Chen, and Mingyu Gao. 2023. ABNDP: Co-optimizing Data Access and Load Balance in Near-Data Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 3–17. doi:10.1145/3582016.3582026

[52] Vijay R Tiwari. 2023. Developments in KD tree and KNN searches. *International Journal of Computer Applications* 975 (2023), 8887.

[53] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[54] Yi Wang, Dun Liu, Hongmei Zhao, Yali Li, Weimeng Song, Menglin Liu, Lei Tian, and Xiaohao Yan. 2022. Rapid citrus harvesting motion planning with pre-harvesting point and quad-tree. *Computers and Electronics in Agriculture* 202 (2022), 107348.

[55] Jingyi Xu, Sehoon Kim, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Memory-Efficient Hardware Performance Counters with Approximate-Counting Algorithms. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 226–228. doi:10.1109/ISPASS51385.2021.00041

[56] Zhou Yijun, Xi Jiadong, and Luo Chen. 2021. A Fast Bi-Directional A* Algorithm Based on Quad-Tree Decomposition and Hierarchical Map. *IEEE Access* 9 (2021), 102877–102885. doi:10.1109/ACCESS.2021.3094854

[57] Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E. Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B. Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, OR, USA) *(SPAA '25)*. Association for Computing Machinery, New York, NY, USA, 350–366. doi:10.1145/3694906.3743318

[58] Yue Zhao, Yunhai Wang, Jian Zhang, Chi-Wing Fu, Mingliang Xu, and Dominik Moritz. 2021. KD-Box: Line-segment-based KD-tree for interactive exploration of large-scale time-series data. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 890–900.